Programozható mérésautomatizálás mikroprocesszor alapú vezérlővel

Mészáros Bence
OE-KVK, FIWWQS
2013

Tartalomjegyzék

1 - Bevezetés

1.1 - A témakör indoklása

A mérés a megismerés eszköze. Jelen van minden szakterületen és a mindennapi életben is. Célja valamilyen „fizikai jelenség” objektív jellemzése. A *mérést* definiálhatjuk, mint műveletek sorozatát. A mérést végző személy feladata a műveleti fázisok tervezése, kiépítése, vezérlése és az eredmények kiértékelése. A mérés megtervezése valamilyen tudományosan elfogadott elvek és relációk alapján történik. A felhasznált eszközök kezelése, a paraméterek állítása, a mérési pontok leolvasása, és a kiértékelés a *mérési folyamat.*

Az *iparban* a mérések sebessége, reprodukálhatósága és pontossága, az eredmény könnyű feldolgozhatósága fő szempontok a mérések kiépítése során. A mérési folyamat éppen ezért túlnyomórészt automatizált ezen a területen. Ha a méréseket kézzel végeznék, drasztikusan megnőne a szükséges idő, rengeteg emberi erőforrást vonna el, illetve megnőne a szubjektív hibák lehetősége is. A mérési pontok felvétele és a kiértékelés igen nehézkes és körülményes lenne. Az egyes mérési folyamatok ember nélküli vezérlésének megvalósítására rengeteg céleszköz áll rendelkezésre, a kommunikáció és az illesztés a részfeladatot ellátó egységek között szabvány alapú, így az egyes mérések kiépítése jóval egyszerűbb. A mérési eredmények feldolgozása és kiértékelése automatikus.

A mérésautomatizálás erősödő tendencia a „kutatás és fejlesztés” területein is. Itt azonban a mérések kisebb száma és talán a költségek ráfordíthatósága miatt, célspecifikusabb megoldásokra lehet szükség. A mérési folyamatokat vezérlő egység (továbbiakban mérésvezérlő) és a műszerek kommunikációja, illesztése, a vezérlést tervező személy feladata, a komplett, könnyen egymáshoz illeszthető modulok hiányában. Ez programozási szaktudást, és specifikus fejlesztői eszközöket követelhet meg.

A szakdolgozat témája egy olyan mérésvezérlő szoftverének megvalósítása, amelyre a mérési vezérlés megírása nem igényel semmilyen külső fejlesztői eszközt, csak egy egyszerű szövegszerkesztőt PC-n, és egy SD kártya olvasót. A vezérlés tehát szöveges alapú. A felhasználó a kommunikációt és az illesztést készen kapja, csak a konkrét mérési problémával kell foglalkoznia. Könnyen érthető parancsokkal tud kommunikálni az eszközökkel, az „alsóbb”, tényleges hardveres és szoftveres megvalósítást nem kell ismernie. A megírt szöveges vezérlést SD kártyán lehet az eszközbe helyezni, majd a készülékről futtatni. A mérési eredmények eltárolása és a kiértékelés az SD kártyára menthető. Az eszközzel jelentősen csökkenthető a mérésekre összességében szükséges idő. Használata alapfokú programozási tudást igényel, ami akár a

dokumentáció áttanulmányozásával elsajátítható, így széles körben használható egyéni igények szerint.

1.2 - A témakör felvezetése

A *szoftver* a „hardver” működésének irányítása, funkcióinak vezérlése egy adott feladat érdekében. A legtöbb szoftvert ma *magas szintű programozási nyelven* írják. A magas szint a gépi kódtól való elvonatkoztatás (absztrakció) mértékére utal. A *gépi kód* olyan utasítások összessége, amelyet a processzor közvetlen végre tud hajtani. Az *utasítás* egy bit sorozat. A *bit* csakis két diszkrét állapot vehet fel. A két állapotot mi legtöbbször 0-val, illetve 1-el jelöljük. A processzor a fizikai felépítéséből kifolyólag értelmezni képes a bitekből felépülő utasítást. Az *utasításkészlet* a processzorhoz használható utasítások listája. Minden processzor architektúra saját utasításkészlettel rendelkezik, az eltérő felépítésükből eredően. Egy architektúrán belül egy konkrét típusnak lehetnek egyedi utasításai is.

A *programozási nyelv* alatt egy olyan formális (szimbólumok és rajtuk definiált szabályokon alapuló) nyelvet értünk, melynek célja hogy a felhasználónak ne egyesekkel és nullákkal kelljen kommunikálnia a processzorral. A legtöbb programozási nyelv szöveg alapú. Magas szintű programozási nyelvek összetett adatstruktúrákkal, összetett aritmetikai és logikai kifejezések kezelésével, illetve egyéb speciális nyelvi elemekkel rendelkeznek a kívánt cél eléréséhez. A processzor számára ezek közvetlen nem értelmezhetők.

A magas szintű programozási nyelven megírt *forráskódot* gépi kóddá kell alakítani, hogy „futtatható” legyen. Ennek két módja ismeretes.

1. A forráskódot a *fordítóprogram* az adott programozási nyelv szabályai alapján elemzi, ha nem talál hibát, gépi kóddá alakítja, így kapjuk a *futtatható állományt*. Fordítóprogram alatt itt azon szoftverek összessége értendő, amelyek által létrejön a futtatható állomány.

2. Az *interpreter* (értelmező) a forráskódot, közvetlen vagy közvetve, részegységenként gépi kóddá alakítja, és futtatja a feldolgozás közben. Ha hibát talál, leáll. Nem jön létre futtatható állomány.

Az interpreter jellegű feldolgozás általában lassabb, mivel az interpreter-nek minden futáskor elemeznie kell a forráskódot, míg a gépi kóddá fordított forráskódnál ez nem szükséges, csak egymás után végre kell hajtani az utasításokat. A sebességbeli különbség széles skálán mozoghat a körülmények függvényében.

Mivel minden architektúra külön utasításkészlettel rendelkezik, ahhoz hogy egy specifikus programnyelven megírt forráskódból futtatható állományt generáljunk, léteznie kell az adott architektúrára fordítóprogramnak. A futtatható állományt a különböző architektúrák között nem hordozhatjuk. A fenti két módszernek léteznek különböző átmenetei. Vannak olyan programnyelvek, amelyekből a fordítóprogram egy „virtuális hardverre" hoz létre futtatható állományt. A virtuális hardver egy szoftver. Az erre létrehozott futtatható állomány hardver független. A futtatható állományok így könnyen hordozhatók a különböző architektúrák között, amelyekre a „virtuális gép" szoftver meg van valósítva.

Beágyazott rendszer alatt olyan hardvert értünk, ami egy szoftvert „futtató" integrált áramkör (például mikrokontroller) köré épülve valósít meg egy cél specifikus feladatot. Az *integrált áramkör* egy kisméretű félvezető lapkán megvalósított áramkör, angolul „Integrated Circuit", röviden *IC*. A szoftver az IC-be van „ágyazva". A *mikrokontroller* egy mikroprocesszor, illetve néhány „periféria" áramkör egy IC-n. Ezt egészítik ki a külső áramkörök és egyéb IC-k, amikkel a mikrokontroller a szoftver segítségével kommunikál a kívánt cél elérése érdekében. A mai gyártás mellett a mikrokontrollerek olcsó megoldást jelentenek vezérlő „intelligencia" eszközbe „ágyazásához". Az eszköz szinte bármi lehet. Egy mérésvezérlő, kávéfőző, mobiltelefon, elektromos fogkefe, gépjármű, stb.. A hardver megvalósítása csak a kívánt cél eléréséhez feltétlen szükséges egységeket tartalmazza.

PC-k esetén a processzor mellett szükség van külső egységre, ahol tároljuk a szoftvert. A szoftver nem egy darab IC-n fut. Lehetőség van igen sok komplex külső eszköz csatlakoztatására, amik mind egyenként valósítanak meg egy cél specifikus feladatot. Merevelemez, hálózati kártya, hangkártya, stb.. A felhasználhatóság éppen ezért rendkívül széles.

1.3 - A feladat megfogalmazása

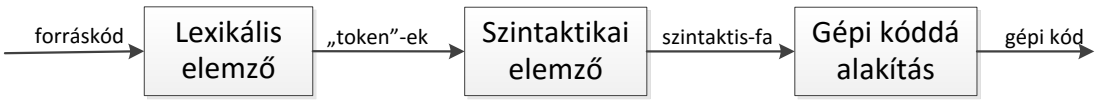
Beágyazott rendszerek esetében a program fejlesztése nem a cél architektúrán zajlik. A fordítóprogram a PC-n fut, viszont a használt mikroprocesszor számára generálja a gépi kódot. Ezt el kell juttatni a mikrokontrollerre. Ehhez specifikus szoftveres és fizikai eszközökre lehet szükség. Ezek nem állnak mindenki rendelkezésére. A cél ennek megkerülése, erre a megoldás adott volt. A vezérlőbe SD kártyán lehessen a PC-n megírt szöveges mérésvezérlést helyezni, majd közvetlen az eszközről futtatni. A kérdés a feldolgozás módja, illetve az esetleges PC-s előfeldolgozás volt. A PC-s előfeldolgozást esetleg indokolhatná a jelentősen nagyobb számítási kapacitás, és a megvalósításhoz használható programnyelvek és segédszoftverek széles választéka. Ehhez azonban programot kéne feltelepíteni az eszköz használatához. Ez céges

környezetben macerás, általában nem lehet csak úgy programokat telepítgetni. Összességében a lehetséges előnyök és hátrányok mellett ez az ötlet el lett vetve. A feldolgozásnak teljes mértékben az eszközön kell történnie.

A feladat tehát a vezérlőbe helyezett SD kártyán található szöveges fájl feldolgozása, ami az általunk definiált programnyelven írt szöveges vezérlést, vagyis a forráskódot tartalmazza.

A mérésvezérlőn egy PIC típusú mikrokontroller található, a feldolgozó szoftvert erre írjuk, C programnyelven. Ehhez a gyártó biztosít fejlesztői környezetet PC-re. A *fejlesztői környezet* többek közt tartalmazza a forráskód szerkesztéséhez szükséges felületet, a mikroprocesszorhoz C fordítóprogramot, és a generált gépi kód eljuttatását a mikrokontrollerre. (Ehhez a PC-t és a mikrokontrollert összekötő külső eszköz szükséges.)

1.4 - Lehetséges megoldások



1.3 – 1. ábra: Forráskód feldolgozásának egyszerűsített menete

Az 1.3.1 ábra szemlélteti egy forráskód feldolgozásának (igen) egyszerűsített menetét.

1. Lexikális elemző: A forrásfájl *tokenizálása*, azaz felbontása a szintaktikai elemző számára értelmezhető szimbólumokra. A szimbólumokat karakterek sorozata reprezentálja.

2. Szintaktikai elemző: A lexikális elemző által generált token-ek sorozata alapján felépíti a *szintaktus-fát.* Ez egy olyan alá-fölérendeltségen alapuló struktúra az adott nyelv szimbólumaival, ami a feldolgozás alatt lévő forráskód „futási” struktúráját reprezentálja.

3. Gépi kóddá alakítás: A szintaktus-fa alapján gépi kód generálása.

A forráskód karakterek sorozata. A lexikális elemző reguláris kifejezések által meghatározott szabályok segítségével bontja a karakterek sorozatát token-ekre. A *reguláris kifejezések* segítségével karakterek között definiálhatunk szabályokat, ezáltal karaktersorozat mintákat. A programnyelv szimbólumainak karaktersorozat minták felelnek meg. Ezek a *token-ek.* Ezek különböző típusai valósítják meg a felhasználható nyelvi elemeket, ezekre épülnek a nyelvi szabályok. Összetett matematikai kifejezések

feldolgozása is elvárás. A szintaktikai elemző a „felcímkézett” token-ek alapján felépíti a szintaktus-fát, ebből generálódik a gépi kód.

A reguláris kifejezések feldolgozását szoftveresen meg kell valósítani. Ezáltal definiálni kell a token-ek lehetséges típusait, azaz lehetséges karaktersorozat mintákat, a programnyelv szabályait, és a matematikai kifejezések szabályait. A szintaktikai elemzőnek valamilyen algoritmus alapján a token-ek sorozatából fel kell tudnia építenie a szintaktus-fát, majd ezt valamilyen algoritmus segítségével gépi kóddá kell tudni alakítani.

Ennek szoftveres megvalósítása, az algoritmusok megtervezése igen bonyolult feladat. Léteznek olyan szoftverek, amik kifejezetten fordítóprogram írására lettek tervezve. Tulajdonképpen egy speciális programnyelvet valósítanak meg, ami programnyelv szabályok leírására lett létrehozva. Egy ilyen programnyelven írt forráskódból egy második szoftver segítségével generálhatunk például C kódot, majd ebből fordítóprogrammal gépi kódot. Kifejezetten mikrokontrollerre létrehozott ilyen nyelvet én nem találtam. Mivel a PC és a mikrokontroller két különböző processzor architektúra, a generált C forráskód támaszkodhat olyan megoldásokra, amik esetleg csak PC-n elérhetők. Mélyebb ismeretek hiányában, az ebből adódó esetleges jövőbeli problémák lehetőségét nem tudtam kizárni.

A fent felsoroltak alapján a megvalósítás elméleti hátterének elsajátításához - úgy ítéltem meg -, jóval több idő kell, mint amennyi rendelkezésre áll. Ez kezdetben nagyjából három hónap volt, majd jelenleg (2013.május) 6 hónapra duzzadt. Mindenképpen egyszerűsíteni kellett a feldolgozás menetét, hogy a rendelkezésre álló idő alatt elérhető legyen a kívánt cél. A fejlesztést Szabados Áron szoftverfejlesztő gyakornoktól vettem át, az ő munkája jelölve van a szakdolgozat folyamán.

1.5 - Választott megoldás

A feldolgozó szoftvert C programnyelven írjuk a mikrokontrollerre. Ez lehetőséget nyújt készen kapott, összetett szövegelemző funkciók használatára. A szöveg tokenizálására is létezik készen kapott C funkció. Ez a megadott elválasztó karakterek alapján szedi szét token-ekre a szöveget. Ezáltal, ha a bemenetet soronként tároljuk, alapozhatjuk az elválasztó karakterekre a tokenizálást. Ezzel ki lehet küszöbölni a karakterenkénti, reguláris kifejezéseken alapuló feldolgozást.

A szöveget *tokenizáljuk*, viszont a token-eket nem címkézzük fel típusonként. A sor első token-éből következtetünk a sor típusára. Az első token-t elválasztó karakterek lehetséges száma véges, ez az általunk definiált nyelvi szintaktusból következik. Szintén következik, hogy a sor első token-e alapján a sor típusa meghatározható, a sor típusok száma véges. A saját nyelvi szintaktust úgy kell definiálni, hogy ez igaz maradjon. A sor típusából a várt lehetséges token-ek alapján, az azokat elválasztó lehetséges

karaktereket ismerjük, ezek kombinációja a sor jellegéből fakadóan szintén véges. Minden lehetséges sor típusra feldolgozó állapotot implementálunk C-ben, ezek generálják közvetve a gépi kódot, ez nem a mi feladatunk. A *C-ben implementáláson*, az általunk PC-n megírt C forráskódból mikrokontrollerre generált szoftvert értjük. Ez végtelen ciklusban „fut” a mikrokontrolleren, figyeli a bemeneteket (pl.: gombok, soros port), és ezekkel vezérelhető.

Matematikai kifejezések feldolgozását továbbra is karakterenként végezzük, de nem reguláris kifejezések segítségével. Ezt egy külön feldolgozó egység valósítja meg. Tudjuk előre, hol ütközhetünk a feldolgozás mentén matematikai kifejezésbe. Ilyenkor „átadjuk” a bemenetet a matematikai kifejezést feldolgozó egységnek, ami „visszatér” egy számértékkel, és a forráskód feldolgozása tovább folytatódik.

2 – Hardver

2.1 - Kialakítás

A hardver Molnár Attila Dénes és Móra Gergely munkája. A megvalósítás részletes ismertetése nem célja a szakdolgozatnak.

Az eszközzel szemben támasztott elvárások a következők voltak:

- kalibrált laborműszerekkel kommunikáció,
- mérési adatok tárolása,
- mérések lebonyolítása önállóan,
- kommunikáció a felhasználóval.

A számításba vett laborműszerek:

- asztali multiméter,
- függvény generátor,
- labortáp,
- műterhelés,
- digitális oszcilloszkóp.

A kommunikáció a műszerekkel legtöbbször soros porton keresztül, vagy analóg jellel történik.

Több eszköz felfűzésére, CAN buszra és LIN buszra fűzhetőség lehetősége is ki van alakítva. A LIN (Local Interconnect Network) busz az autóiparban elterjedt kommunikációs csatorna.

A mérési adatokat SD kártyán vagy SRAM-on tárolhatjuk. Egy RTC (valós-idejű óra) IC is található a mérésvezérlőn a mérési idők regisztrálásához.

Az önálló mérések lebonyolításához használható egy földhöz referált 12 csatornás analóg-digitális átalakító, egy digitális-analóg átalakító, és 16 kimeneti/bemeneti csatorna 5V jelszintekkel.

Az eszközön található egy LCD kijelző és 5 nyomógomb. PC-s terminálról, soros porton is kommunikálhatunk. Ennek célja a mérésvezérlés kiválasztása, illetve rövid üzenetek a felhasználó felé.

A jelenlegi kialakítása két nyomtatott áramköri lapból áll, ezeket két külön mikrokontroller kezeli.

- Vezérlő panel,
- kijelző panel.

A vezérlő panel 5-20V-ról működtethető. Innen kapja a kijelző panel is a tápellátást. Az eszköz kialakítása többféle belső szűrést, pufferelést és stabilizálást tartalmaz, az IC-k és a kommunikációs vonalak védelmében. A hardveres USB támogatásért egy harmadik mikrokontroller felelős. Ez a PC felől kapja a tápellátást.

A vezérlő panelen a mikrokontroller és a külső IC-k között a kommunikáció SPI vagy I2C szabvány alapú buszon történik, amit a mikrokontroller vezérel.

BUSZ	**„felfűzött" IC-k**
SPI1	SRAM, SD kártya
SPI2	A/D és D/A átalakító
I2C	RTC, EEPROM, kijelző panel mikrokontrollere

2.1.1 - 1. ábra: Vezérlő panel kommunikáció

A vezérlő panelon egy dsPIC33FJ256MC710A [1] típusú mikrokontroller található. Ez egy digitális jel feldolgozására optimalizált, 16-bites, 40 MIPS (millió utasítás per másodperc) sebességű mikroprocesszort és néhány periféria áramkört tartalmaz.

A kijelző panelt egy PIC16F1938 [2] típusú 8-bites, 8 MIPS sebességű mikrokontroller vezérli. Ez tartalmaz egy integrált LCD vezérlőt. A vezérlőpanelen lévő mikrokontrollerrel I2C buszon össze van kötve.

A mérésvezérlő USB portját egy PIC18F2550 [3] mikrokontroller kezeli. Ez direkt erre kialakított és ajánlott a gyártó által.

2.4 - Jelenleg tervezett változtatások

Szoftveresen gondok voltak a SD kártya és az SRAM-mal való kommunikációk során. Kiderült, az SD kártya nem felel meg 100%-ban az SPI szabványnak. A buszt vagy védeni kell a nem várt időben - azaz amikor az SD kártya nincs kiválasztva kommunikációra - érkező adatoktól, vagy külön buszra rakni. A mérésvezérlő következő verzióján az SD kártya külön SPI buszon lesz.

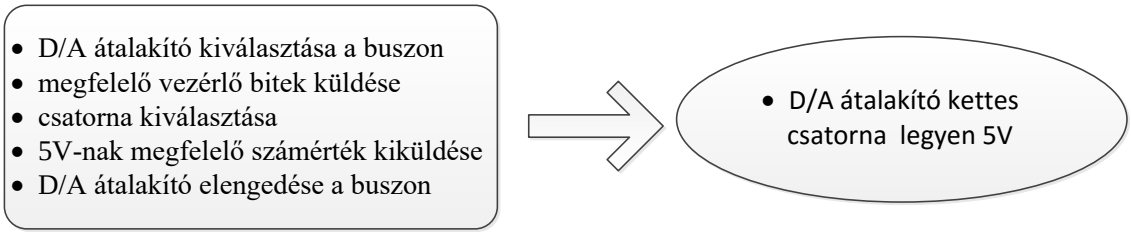
A vezérlő panelen található mikrokontroller egy több lábú, hasonló paraméterekkel rendelkező mikrokontrollerre lesz kicserélve. Ennek oka, hogy a kijelzés kezelésére ne

kelljen külön mikrokontroller, illetve hogy a jelenlegi mikrokontroller csak két SPI vonalat támogat.

3 - Szoftver

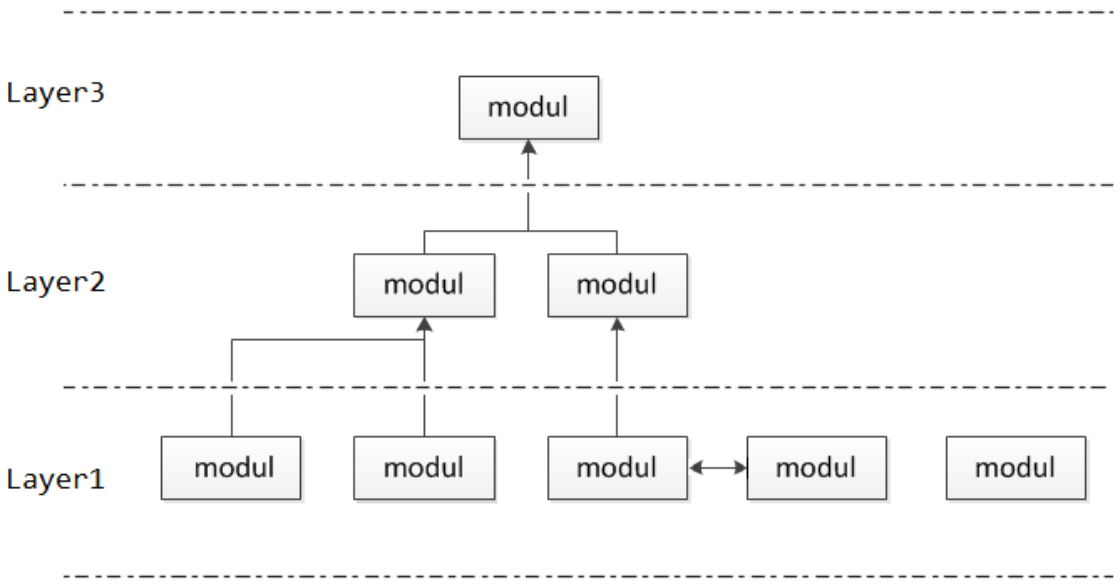
3.1 - Megvalósítás módja

A szoftveres probléma megoldását különálló, részfeladatköröket megvalósító egységekre osztjuk. Az egy részfeladatkör megvalósítását végző egységet nevezzük *modulnak*. Minden modulnak csak a saját feladatkörét befolyásoló információkat kell lekezelnie. A modul belső működéséhez szükséges információk a külvilág felé nem láthatók. A modulok egymásra épülnek, ezáltal egyfajta elvonatkoztatást, absztrakciót tesznek lehetővé a teljes probléma minden „részletétől". A modulok egymás funkcióit felhasználva, valamilyen egyre összetettebb elvonatkoztatást (absztrakciót) valósítanak meg.



3.1 – 1. ábra: Szoftveres absztrakció

A modulok a megvalósított absztrakció mértékéhez viszonyítva különböző vertikális szinteken helyezkednek el. Nevezzük ezeket a szinteket *layer*-eknek. Egy modul általában a közvetlen alatta és felette lévő layer-en található modulokkal kommunikál, az egy szinttel alatta lévő modulokat használja fel működéséhez, felette más modulok épülhetnek a kimenetére.



3.1 – 2. ábra: Modulok struktúrája

Ennek célja az információ struktúra alapú egységbezárása. Ez megkönnyíti a tervezést, az esetleges hibák lokalizálását, újrafelhasználhatóvá teszi a modulokat, és szétoszthatóvá teszi a fejlesztést több programozó között.

A modulok egymás közötti kommunikációja minél jobban megtervezett, annál jobban átlátható és fejleszthető a teljes rendszer. A kommunikációt a 3.1. – 2. ábrán nyilak jelzik. A modulok közti kommunikációt a használt *változók láthatósága,* illetve *paraméterek átadása* valósítja meg. A *változók* valamilyen adattípust (információt) tárolnak, a modul lehetséges paraméteri a formálisan definiált, „átadható" adattípusok.

A modulokat *blokkok* építik fel. Ezek adatstruktúrákat, és rajtuk végzett algoritmusokat írnak le az adott programnyelven. *Modulra lokális változó,* csak az adott modul számára elérhető, a modulon kívülről nem látható. A *globális változó* az összes modul számára elérhető. *Blokkra lokális változó* csak az adott blokk számára elérhető.

Egy *változó élettartama* vagy egy blokk lefutásának ideje, vagy a teljes program lefutásának ideje. Ha ez nem kielégítő, dinamikus memóriakezeléssel az élettartam változóként tetszőlegesen vezérelhető.

A szoftver megvalósítása ANSI C programnyelven történt. A modulok esetünkben a *különálló forrásfájlok*, amiket a C nyelv által kínált, nyelvi elemeket megvalósító blokkok alkotnak. A *függvény* egy hivatkozási név által elérhető kódrészlet, valamilyen specifikus feladat ellátására. Paraméterek átadása a függvényeken keresztül történik. A *modulra lokális függvény* a többi modul számára nem látható, a modul belső működéséhez szükséges funkciót valósít meg. A modul g*lobális függvényei* a többi modul számára elérhetőek, a modul feladatkörének megfelelő funkciókat valósítanak meg, a modulok közti kommunikáció ezeken keresztül történik.

3.2 - Fejlesztői eszközök

A fejlesztés MPLAB X IDE v1.51 [4] fejlesztői környezetben, C30 v3.30b [5] fordítóprogrammal, ICD3 [6] hardveres programozóval és debugger-rel történt.

3.3 - Funkcionális elvárások

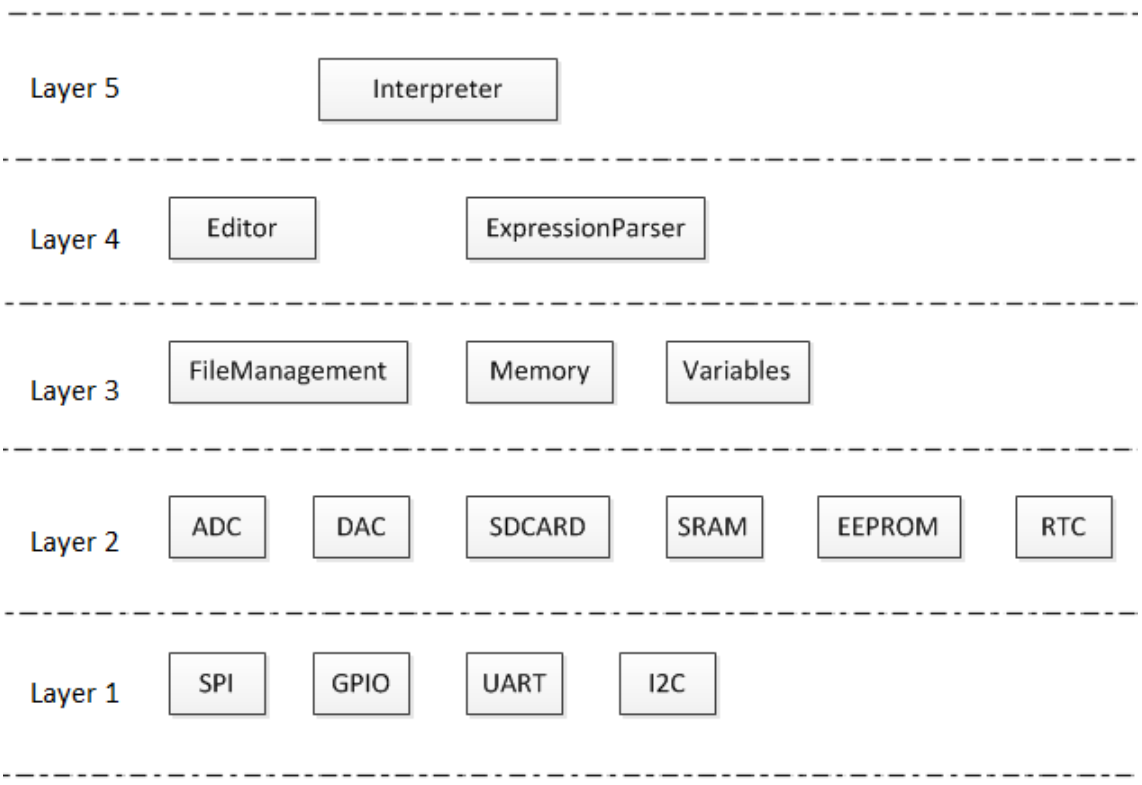
Az eszköz célja mérési folyamatok automatizálása és az eredmények tárolása. Elvárás az egyszerű, specifikus külső eszközt nem igénylő vezérelhetőség. Ez szöveges formában történik, a megírt mérést lehetőség van SD kártyáról fájlként, vagy soros porton, PC-s terminálon keresztül bevinni, illetve menteni. A szoftver működésének funkcionális elvárásai:

- SD kártyán fájlból mérésvezérlés futtatása, a kijelző és a gombok segítségével,

- PC-s terminálon soros porton keresztül az SD kártyára mérésvezérlés bevitele és szerkesztése,
- a mérésvezérlésen belül változó deklarálás, értékadás,
- matematikai műveletek értelmezése (aritmetikai operátorok, relációs operátorok, logikai operátorok, zárójelezés),
- feltételes elágazások és ciklusok értelmezése,
- kimenet mentése a kívánt helyre (pl. fájl, soros port),
- perifériák „magas szintű” elérése és kezelése,
- hiba esetén könnyen értelmezhető üzenet a felhasználó felé

3.4 - Megvalósítás menete

A megvalósítandó modulokat és felosztásukat a 3.4 - 1. ábra szemlélteti.



3.4 – 1. ábra: A megvalósítandó modulok és struktúrájuk

A külső hardveres perifériákat közvetlen érjük el. A perifériák I2C, vagy SPI szabványt megvalósító adatbuszra vannak „felfűzve”. Az adatbuszokon a kommunikációt a szoftver vezérli. Ezt valósítja meg a legalsó layer, plusz a soros port és az általános célú

ki és bemenetek kezelését. A második szinten a külső perifériák funkcióit kezelő modulok találhatók. Ezekre épülnek a mérési vezérlés tárolását, a mérési vezérlésben használt változók kezelését, és az SD kártya fájlkezelését megvalósító modulok. A kifejezés feldolgozó modul (ExpressionParser), és a szerkesztő (Editor) modul található a legfelső szint alatt, a legfelső szinten pedig az Interpreter modul.

- *Szabados Áron által fejlesztett, átvett modulok:* SPI, I2C, UART, GPIO, SDCARD,
- *általam továbbfejlesztett modulok:* ADC, DAC, SRAM, Variables, RTC, EEPROM,
- *saját modulok:* Memory, Editor, ExpressionParser, Interpreter.

A modulok rövid leírása és függőségeik:

- *GPIO:*
  Általános célú ki és bemenetek kezelése.
  Függőségek: nincs
- *SPI, I2C, URAT:*
  A megfelelő szabvány szerinti kommunikációt valósítják meg.
  Függőségek: nincs
- *ADC, DAC:*
  Az analóg-digitális átalakító és a digitális-analóg átalakító kezelése.
  Függőségek: SPI
- *SDCARD:*
  SD kártya alacsonyszintű (fájlrendszer nélküli) kezelése.
  Függőségek: SPI
- *SRAM:*
  A külső SRAM írása, olvasása.
  Függőségek: SPI
- *EEPROM:*
  A külső EEPROM írása, olvasása.
  Függőségek: I2C
- *RTC:*
  A „valós-idejű óra” kezelése.
  Függőségek: I2C
- *Memory:*
  A mérési vezérlés kódjának tárolása és soronkénti kezelése.
  Függőségek: SRAM
- *Variables:*
  A mérési vezérlés által létrehozott változók kezelése.
  Függőségek: SRAM

- *FileManagement:*
  A SD kártya magas szintű (fájlrendszeres) kezelését valósítja meg.
  Függőségek: SDCARD
- *Editor:*
  PC-s terminálon soros porton keresztül az SD kártyára mérésvezérlés bevitele és szerkesztése.
  Függőségek: UART, Memory
- *ExpressionParser:*
  Matematikai kifejezés feldolgozó modul.
  Függőségek: nincs
- *Interpreter:*
  A mérési vezérlés feldolgozása.
  Függőségek: Memory, Variables, ExpressionParser

3.5 - Az Interpreter modul

3.5.1 - Funkcionális specifikáció

Elvárások a modullal kapcsolatban:

- a méréshez használt perifériák kezelése a szöveges mérésvezérlés alapján,
- a szöveges mérésvezérlésen belül változó deklarálás, értékadás,
- matematikai műveletek értelmezése (aritmetikai operátorok, relációs operátorok, logikai operátorok, zárójelezés),
- feltételes elágazások és ciklusok értelmezése,
- kimenet mentése a kívánt helyre (pl. fájl, soros port),
- hiba esetén könnyen értelmezhető üzenet a felhasználó felé.

A megvalósítandó nyelv szintaktikáját a következő fejezet (3.5.2) ismerteti.

3.5.2 - Saját nyelv szintaktikája

A szöveges vezérlés saját nyelvi szintaktikával (nyelvi szabályokkal) rendelkezik. A nyelvi szintaktikai egyszerű, minimális programozási tudást követel meg. A sorokat az új sor karakter választja el.

A *kifejezés* számértéken, változókon, és az elérhető C-ben implementált, - visszatérési értékkel rendelkező – függvényeken, a támogatott operátorokkal műveletek végezése. A kifejezésben bármilyen mély zárójelezés támogatott. A kifejezés hamis, ha számértéke nulla, igaz, ha számértéke nullától eltérő.

A nyelvi elemek a következők:

Változók, adattípusok

Név	Típus	Létrehozás	Példa
Variable	Számérték.	*var*	*var valtozo_neve*
Array	Egy dimenziós változó tömb.	*dim*	*dim tomb_neve[10]*
String	Szöveg.	*str*	**még nem implementált*

A változók nevében használható karakterek: angol abc betűi, egészszámok, és az aláhúzás karakter. Minden változót külön sorban kell deklarálni. Tizedespontos számábrázolás támogatott, azonban nincs külön adattípus erre.

Ha egy változónak nem adunk értéket deklaráláskor, az értéke nulla. Deklaráláskor csak közvetlen értéket adhatunk, kifejezéssel nem tehetjük egyenlővé a változót, ezt a deklarálástól külön sorban lehet megtenni.

Tömb tagjának csak a deklarálástól külön sorban adhatunk értéket, alapból minden tag érték nulla. A tömb indexelése egytől kezdődik. A tömb egy tagja vagy a „tomb_neve[index]” formában érhető el, ahol index a kívánt elem sorszáma, - ez bármilyen egészszám értékű kifejezés is lehet-, vagy a „tomb_neveindex” formában, ahol index számérték.

```
var alma = 8.12
var korte = 7
korte = alma + korte * 3.2
dim tomb_egy[10]
dim tomb_ketto[alma+4]
tomb1[3] = 12
tomb_ketto3 = 88
```

Operátorok

Értékadás (=)

Az értékadó operátor egy számértéket, vagy egy kifejezést rendel a változóhoz.

```
alma = 3
alma = alma*7*(korte+2.34)
korte = getadc(2)
```

Ahol „getadc(2)” egy elérhető, C-ben implementált - visszatérési értékkel rendelkező – függvény**.**

*Aritmetikai operátorok (+, -, *, /)*

+	összeadás
-	kivonás
*	szorzás
/	osztás

Az aritmetikai operátorok megegyeznek a matematikában használtakkal.

```
alma = 7+7*4.3/2-1000
```

Relációs operátorok (==, !=, >, <, >=, <=)

==	egyenlő
!=	nem egyenlő
>	nagyobb mint
<	kisebb mint
>=	nagyobb egyenlő mint
<=	kisebb egyenlő mint

Két kifejezés értékének összehasonlítására az alábbi operátorok támogatottak. A kifejezés eredménye vagy igaz (1), vagy hamis (0).

Logikai operátorok (!, &&, ||)

A logikai operátorok Boolean-algebrai kifejezéseknek felelnek meg. Egy Boolean kifejezés eredménye vagy igaz (nem nulla, 1-el jelölve), vagy hamis (nulla). A logikai operátorok jelenleg (2013. május) maximum két kifejezésen használhatók.

a	!a
0	1
1	0

A **!** operátornak csak egy operandusa van, az operátor jobb oldalán, és Boolean logikai tagadást (NOT) valósít meg. Az *operandus* vagy számérték, vagy kifejezés.

a	b	a && b
0	0	0
0	1	0
1	0	0
1	1	1

Az **&&** operátor a Boolean logikai és-nek (AND) felel meg. Legalább két operandusa van, és csak akkor igaz, ha az összes operandus igaz.

a	b	a || b
0	0	0
0	1	1
1	0	1
1	1	1

A **||** operátor a Boolean logiai vagy-nak (OR) felel meg. Legalább két operandusa van, és akkor igaz, ha legalább az egyik operandus igaz.

Feltételes elágazások és ciklusok

Egy program „futása” általában nem korlátozódik az utasítások lineáris végrehajtására. A futás egy feltételtől függően elágazhat, vagy egy utasításblokk megismétlődhet. Az *utasításblokk* elejét és végét nyelvi elemek jelzik.

Az if-else szerkezet

Az if-else szerkezettel lehetőség van feltétel alapú elágazásokat létrehozni.

```
if(kifejezés)
…
else
…
endif
```

Ha a kifejezés igaz, csak az **if** és **else** közti utasításblokk hajtódik végre **endif**-ig. Ha a kifejezés hamis, csak az **else** és az **endif** közti utasításblokk. Az else-ág nem kötelező. A szerkezetet az **if** nyitja és **endif** zárja. Több if-else szerkezet tetszőleges mélységben egymásba ágyazható. Az **else**-el egy sorban nincs lehetőség **if** használatára.

A while szerkezet

Forráskód feltétel alapú ismétlésére alkalmazható a while-szerkezet. Ezt a **while** nyitja és az **endwhile** zárja.

```
while(kifejezés)
…
…
endwhile
```

Ha kifejezés igaz, a while-szerkezetbe zárt utasításblokk végrehajtódik, majd a feldolgozás visszaugrik a **while** sorra, ahol ismét kiértékelődik a kifejezést. Az utasításblokk annyiszor fut le egymás után, amíg a kifejezés értéke nem lesz hamis.

A kimenet mentése

A kimenet mentése a „printf” paranccsal történik. Ennek szintaktikája:

```
printf(paraméterek)
```

A *paraméterek* a parancsnak átadott információ. A **printf** három paramétert vár:

1. PORT lehet: COM0, COM1, COM2, LCD, CAN, LIN
2. idézőjel közé zárt szöveg
3. az esetlegesen szövegbe helyettesítendő változók nevek

```
printf(PORT, "szoveg %var szoveg %var", valtozo1, tomb[2])
```

A második rész kerül majd az első paraméter által meghatározott kimenetre, miután a **%var** címszavak felcserélődnek a harmadik részben felsorolt változók értékeivel. A felcserélés a változók sorrendjében történik.

Beépített függvények

A beépített függvények segítségével érhetjük el a mikrokontroller köré kiépített külső áramkörök és IC-k által megvalósított felhasználható funkciókat.

Ezek megvalósítása még nem teljes. A szoftverben ki van építve már a megoldás, amihez egy lista segítségével és pár sornyi kóddal bármilyen C-ben implementált függvény hozzáadható.

3.5.3 - Az állapotgép megvalósítása

A feldolgozandó sorokat a Memory modul tárolja a külső SRAM-on. A sorok között lépkedni tudunk, adott sorról másolatot kérhetünk, illetve lekérhetjük az összesen tárolt sorok számát. A változókat a Variables modul kezeli, és szintén a külső a SRAM-on vannak tárolva. A kifejezéseket az ExpressionParser modul értékeli ki.

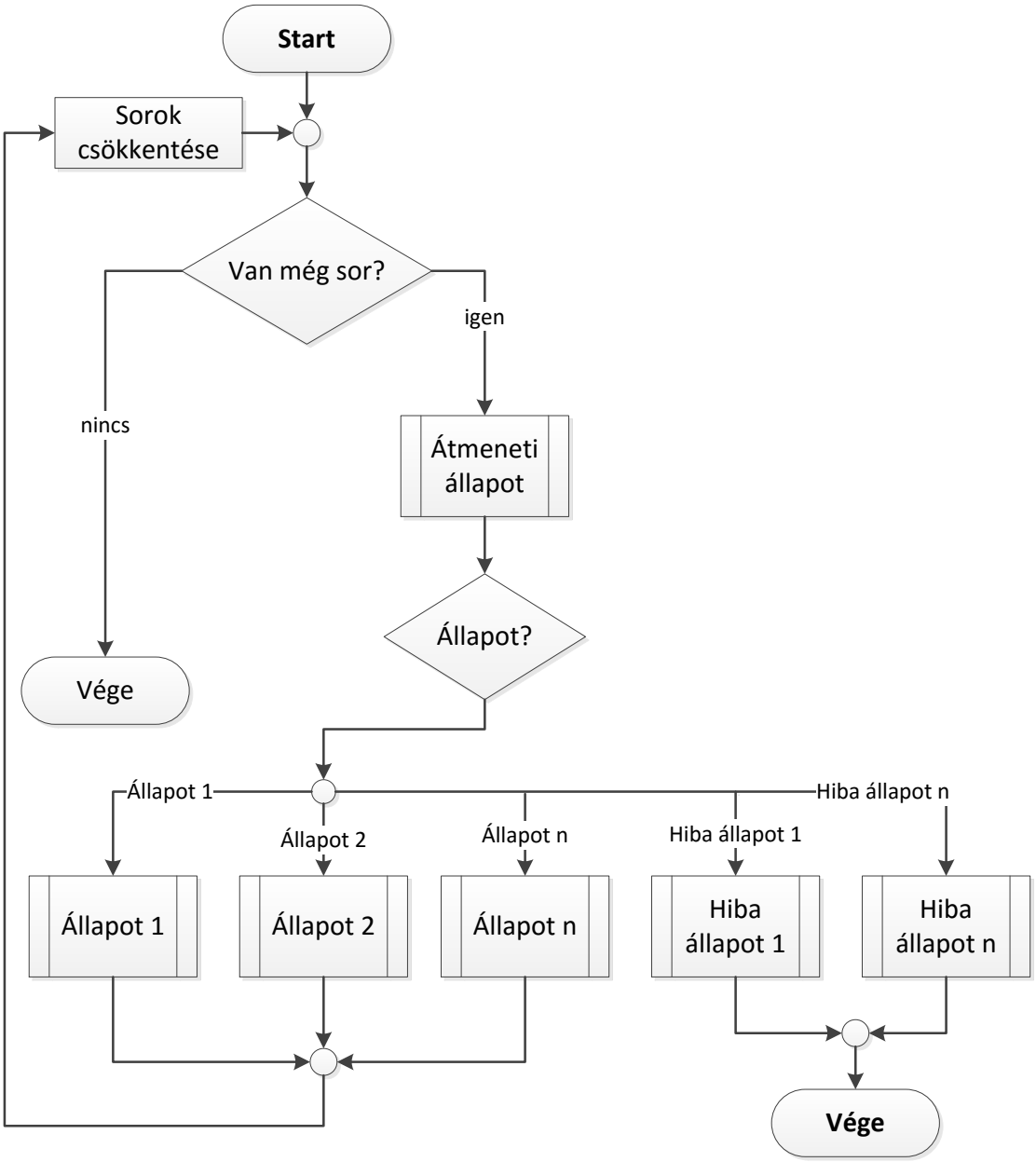
A bemenetet soronként dolgozzuk fel. A feldolgozást egy *bemenet vezérelt állapotgép* valósítja meg. Az állapotgép a bemenet függvényében előre definiált állapotokat vehet fel, azaz a program futásában egy több irányú feltételes elágazást valósít meg. Esetünkben a bemenet a feldolgozás alatt álló sor első token-e.

Minden állapot egy specifikus sor típus feldolgozási módjának felel meg. Nevezzük ezeket *feldolgozó állapotoknak*. Számuk véges. A teljes feldolgozás egyszerűsített menete:

1. feltöltött memória sorok számának lekérése és eltárolása,
2. aktuális sorról lokális másolat készítése,

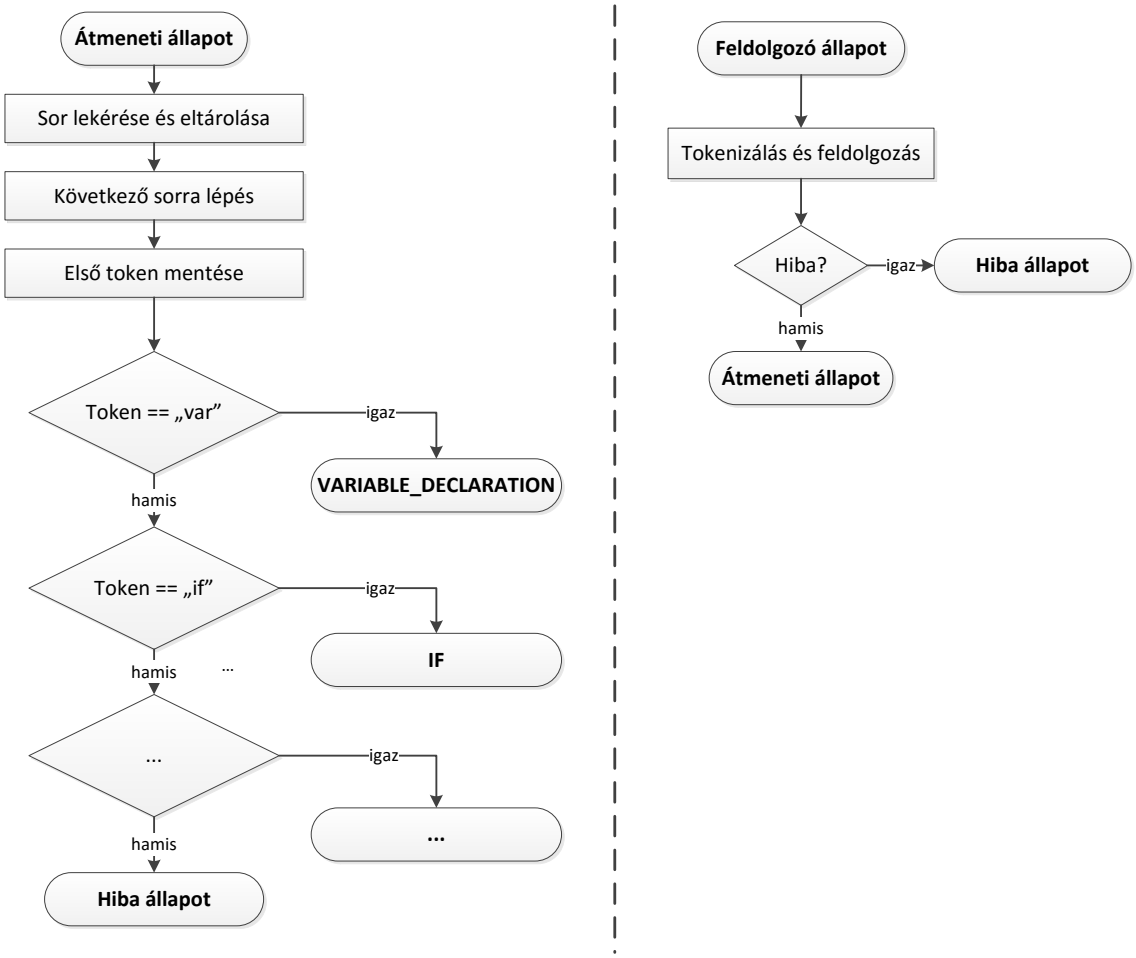
3. ha van még sor, a következő sorra lépés
4. a lokálisan tárolt sor első token-e alapján, a megfelelő állapot kiválasztása,
5. állapot végrehajtása, sorok számának csökkentése, ha van még sor, ugrás a 2. pontra.

Az 1. pont minden futáskor egyszer hajtódik végre. A 2., 3. és 4. pont választja el a feldolgozó állapotokat. Ezt a három műveletet szintén egy állapotnak definiáljuk, ami minden feldolgozó állapot előtt végrehajtódik, tehát minden második állapot ez az *átmeneti állapot* lesz, illetve ez lesz mindig a legelső állapot. A sorok számát egy függvényre lokális változóban tároljuk.



3.5.3 – 1. ábra: Interpreter modul folyamatábrája

Az állapotgépet egy switch-case *szerkezet* valósítja meg. A *switch-case szerkezet* egy integer (egészszám) típusú változó értéke alapján a program futásában több irányú feltételes elágazást tesz lehetővé. A figyelt változó esetünkben az előre definiált állapotoknak megfelelő számértékeket veheti fel. Ha a számértéknek nem felel meg állapot, a switch-case szerkezet alapelméretezett ága fut le, ami ismeretlen hiba állapotba állítja az állapotgépet. A változót az állapotok visszatérési értéke állítja, minden sikeresen lefutott feldolgozó állapot az átmeneti állapotnak megfelelő számértékkel tér vissza a változóba, hiba esetén hiba állapotnak megfelelő számértékkel. Az átmeneti állapot a következő feldolgozó állapottal tér vissza. Így a feldolgozás folytonos.



3.5.3 – 2. ábra: Átmeneti állapot és feldolgozó állapot folyamatábrája

Minden állapotnak egy számérték felel meg, minden számértékhez tartozik egy címke. Ezt az *enum* nyelvi elemmel tudjuk elérni. Konstans számértékekhez rendelhetünk szöveget. A létrehozott változó csakis az előre definiált címkéknek megfelelő

számértékeket veheti fel, egyéb esetben értéke nem meghatározott. Így sokkal könnyebben olvasható lesz a forráskód.

```
// Lehetséges állapotok
enum states_t {
      TRANSITION,                   // = 0
      VARIABLE_DECLARATION,         // = 1
      ARRAY_DECLARATION,            // = 2 stb.
      IF,
      // ...
      ERROR_IF_NOEND,
      // ...
      ERROR_UNEXPECTED_TOKEN
      }

// Állapot feldolgozó függvények
static enum states_t STATE_TRANSITION(void);
static enum states_t STATE_VARIABLE_DECLARATION(void);
// ...
static enum states_t ERROR_STATE_FCIFNOEND(void);
static enum states_t ERROR_STATE_UNEXPECTED_TOKEN(void);
// ...

// A következő állapotot tároló változó
enum states_t Next_State = STATE_TRANSITION;

while(memoryline_counter){          // memoryline_counter -> hátralévő
                                    // sorok száma
        switch (Next_State){
           // Átmeneti állapot
           case TRANSITION:
               Next_State = STATE_TRANSITION();
               memoryline_counter--;
               break;
           // Feldolgozó állapotok
           case VARIABLE_DECLARATION:
                Next_State = STATE_VARIABLE_DECLARATION();
                memoryline_counter--;
                break;
         // ...
         // Hiba állapotok
           case ERROR_IF_NOEND:
               Next_State = ERROR_STATE_FCIFNOEND();
               memoryline_counter--;
                break;
           // ...
           default:
                Next_State = ERROR_STATE_UNEXPECTED_TOKEN();
        }
```

3.5.3. – 3. ábra: Az állapotgép megvalósítása C programnyelven

Az állapotgép könnyen bővíthető. Új állapot felvitele három lépés:

1. a lehetséges állapotokat leíró enum bővítése az új állapot címkéjével,
2. a switch-case szerkezet bővítése az új címkével a többi ághoz hasonlóan,
3. az új állapotot megvalósító függvény megírása.

Az eddig megvalósított állapotok:

Átmeneti állapot és feldolgozó állapotok	
Név	**Rövid leírás**
TRANSITION	Átmeneti állapot.
VAR_DECLARATION	Változó létrehozása.
VAR_SET_VALUE	Változó értékadás.
ARRAY_DECLARATION	Tömb létrehozása.
ARRAY_SET_VALUE	Tömb elemének értékének állítása.
IF	If-else szerkezet feldolgozása.
ENDIF	Csak igaz IF állapot után futhat le.
ELSE	Csak hamis IF állapot után futhat le.
WHILE	While-szerkezet feldolgozása.
END_WHILE	Csak igaz WHILE után futhat le
PRINTF	Kimenetre mentést feldolgozó állapot.
Hiba állapotok	
Név	**Rövid leírás**
ERROR_UNEXPTECTEDT_TOKEN	Nem várt token.
ERROR_VAR_LONG_NAME	Túl hosszú változó név.
ERROR_VAR_DUPLICATE	Már létező változó név.
ERROR_VAR_NOT_FOUND	Keresett változó nem létezik.
ERROR_SRAM_FULL	Megtelt az SRAM. (Változók itt tárolódnak)
ERROR_IF_NOEND	Lezáratlan if-szerkezet.
ERROR_WHILE_NOEND	Lezáratlan while-szerkezet.

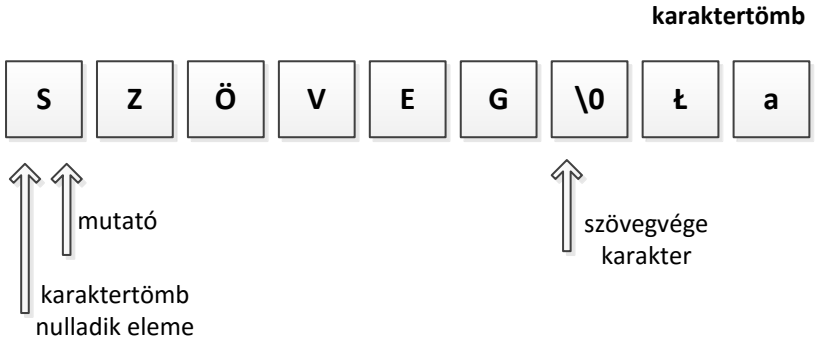
3.5.3.- 4. ábra: Megvalósított állapotok (2013.05.)

Állapotgép megvalósítására léteznek switch-case szerkezeten kívül egyéb megoldások. A switch-case szerkezet mellett az átláthatósága döntött.

3.5.4 - Sorok feldolgozása

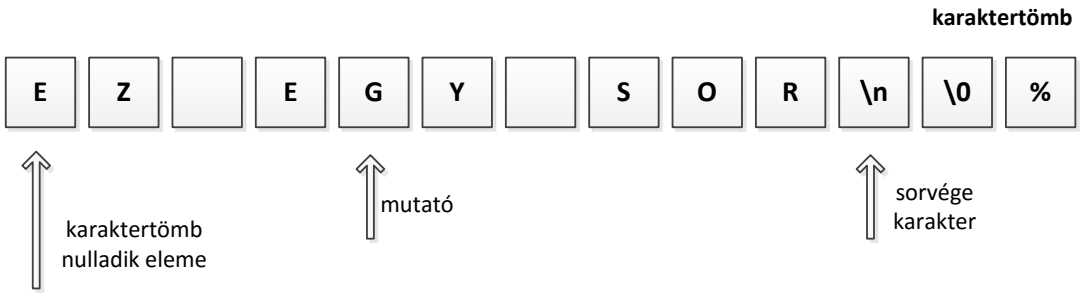
A C nyelvben szöveg tárolására nincs külön adattípus. A *karakter adattípusú* változóban egy darab karaktert tudunk tárolni. Az *egydimenziós tömb* azonos adattípusú változók meghatározott elemű sorozata. Szöveget karaktertömbként lehet tárolni. Ezt elérhetjük karakterenként egy karakter adattípusú *mutatóval*. A mutató értéke egy tárolási cím. Ha eggyel növeljük a mutató értékét, az az adattípusának megfelelő mértékű címnövekedéssel jár együtt. Így egy mutatóval karakterenként lépkedni tudunk a szövegben.

ASCII karakterkódolásnál a karaktereket számok reprezentálják. A szövegvége karakter jelzi egy karaktertömbben a szöveg végét.



3.5.4 – 1. ábra: Szöveg kezelése

A mérési vezérlést a sorvége karakter tagolja *sorokra*.



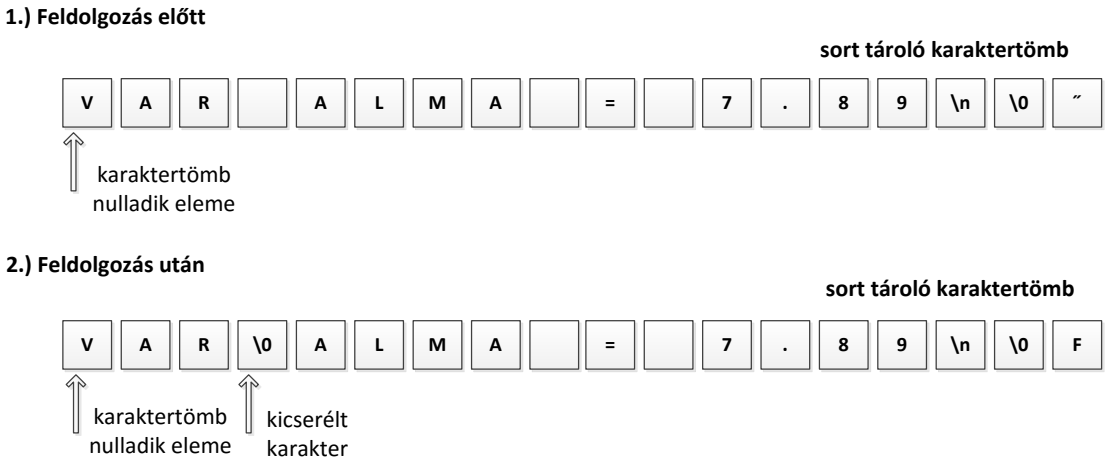
3.5.4 – 2. ábra: Sor kezelése

Ha nem megszabott egy sor maximális hosszúsága, nem tudjuk, mekkora karaktertömbre lehet szükségünk a tárolásához. Ebben az esetben csak karakterenként tudjuk feldolgozni a bemenetet. Ez rengeteg bonyodalmat szül. Éppen ezért definiálva van a maximális sor hosszúság. A Memory modul ekkora karaktertömbökként tárolja a mérésvezérlés sorait. A sor maximális mérete a forráskódban könnyen, egy érték megváltoztatásával állítható. Ez egy erős megszorítás a felhasználó felé, viszont a jelenlegi sor mérettel - ami a várható felhasználás mellett egy pozitív irányba igen túlbecsült érték - több száz sor tárolására van lehetőség. Pazarló megoldás, de a feldolgozás megvalósítását nagyban segíti.

A feldolgozandó sorról egy Interpreter modulra lokális karaktertömbbe másolatot készítünk. A feldolgozás módosítja a karaktertömböt, a közvetlen a Memory modul által tárolt sorokat nem módosítjuk.

Az ANSI C szabvány karaktertömbökön végzett szövegkezelésre egy beépített szövegkezelő modult biztosít. A modul által felhasználható függvények listája, paraméterei, és leírásuk a szabvány dokumentációban illetve az interneten elérhető. Lehetőség van többek között karaktert vagy karaktereket keresni, összehasonlítani, áthelyezni, másolni, kivágni, és tokenizálni is.

Ezt az strtok() függvénnyel tehetjük meg. Az strtok() függvénynek átadjuk a karaktertömb első elemének címét, és a token-t elválasztható karaktert vagy karaktereket. A függvény módosítja a karaktertömböt, és visszatér a token első karakterének címével, ha nem talált token-t, az átadott címmel.

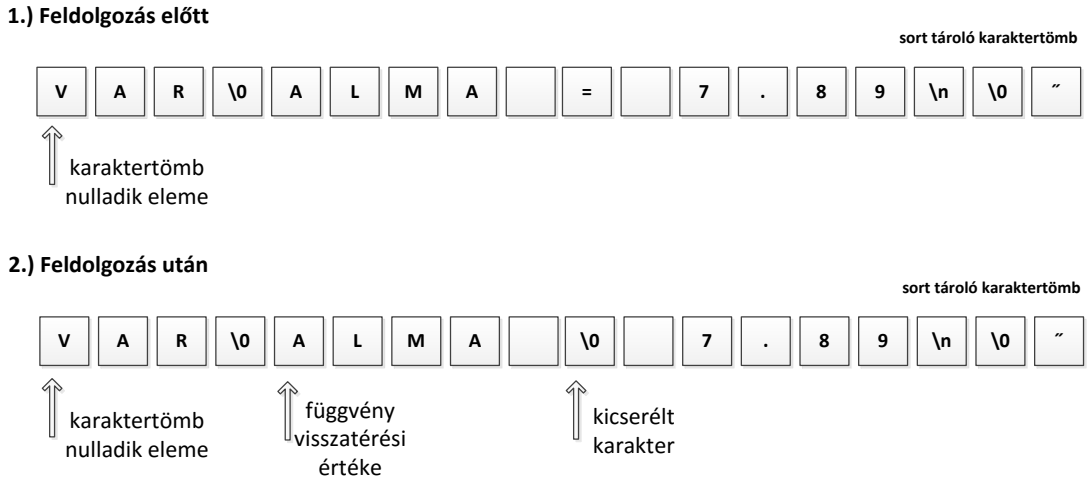


3.5.4 – 3. ábra: strtok() függvény működése

Az strtok() függvény az elválasztó karaktert felcseréli a szövegvége karakterrel. A 3.5.4 – 3. ábrán a szóközkarakter az elválasztó karakter.

A függvény visszatér a karaktertömb nulladik elemének címével. A karaktertömb most a token-nek megfelelő szövegrészt tartalmazza. A szövegvége karakter csak számunkra információ, a következő karakterek nem vesztik el értéküket a karaktertömbben.

Az strtok() függvény egy változóban tárolja a felcserélt karakter címét. A változó élettartalma a program futásának idejére terjed, ezért az értékét megtartja a függvény lefutása után. Következő hívásnál ebből számolja a függvény a következő token első karakterének címét. Ezért második lefutásnál nem adunk át a feldolgozási helyet, csak egy null értékű mutatót, és a következő token-t elválasztó karaktert vagy karaktereket.



3.5.4 – 3. ábra: Következő token „kinyerése”

A 3.5.4 – 3. ábra a sort tároló karaktertömb alakulását szemlélteti. Elválasztó karakter az egyenlőség jel.

A sort tároló karaktertömbünkben még mindig az első token van. Szöveg másolását biztosító függvény az átadott karakter címétől a szövegvége karakterig másolja a karaktereket karaktertömbök között. Token tárolására külön karaktertömböt hozunk létre. Ide az strtok() által visszatért cím segítségével mindig átmásoljuk az aktuális tokent. Ezt megtehetjük, mert a token-t zárja a kicserélt szövegvége karakter.

A sor első token-ét több esetben a szóközkarakter határolja. Ezért ez az egyik első tokent vizsgáló elválasztó karakter. Első token-t határolható karakterek a jelenleg megvalósított saját nyelvi szintaktika alapján:

- szóközkarakter
- nyitózárójel
- egyenlőségjel
- nyitó szögletes zárójel

Lehetséges első token-ek	Elválasztó karakter
var	szóközkarakter
dim	szóközkarakter
if	nyitózárójel
endif	nincs
else	nincs
while	nyitózárójel
endwhile	nincs
változónév	egyenlőségjel
tömbnév	nyitó szögletes zárójel

függvénynév	nyitózárójel
printf	nyitózárójel

A 3.5.4 – 4. ábra: Lehetséges első token-ek és elválasztó karakterek

Az átmeneti állapot ezeknek a szövegrészeknek és az első token összehasonlításával állítja be a megfelelő feldolgozó állapotot.

Az 3.5.4 – 3. és 4. ábrák a mérésvezérlés egy sorának feldolgozási menetét szemléltették. A sor:

```
var alma = 7.89
```

Tokenek	Szöveg és elválasztó karakter	
token1	„var„	szóközkarakter
token2	„alma „	egyenlőségjel

3.5.4 – 5. ábra: A kinyert token-ek

Az token1 alapján a változó deklarálás állapotra lép az állapotgép. A token2-t mindig egyenlőségjel választja el, és a létrehozandó változó nevét tárolja. Ha ebben szóközkaraktert találunk, felcseréljük szövegvégét jelző karakterrel. Változónévben nem lehet szóközkarakter. A token2 után mindig egy számérték van. Ez az strtod() szövegszerkesztő függvénnyel kinyerhető, ami visszatér a karaktereknek megfelelő számértékkel. A tizedespont támogatott, a szóköz karaktert kihagyja, ha vége a számnak leáll. Ha nem talál számot, nullával tér vissza. A számot egy változóba mentjük. A Variables modul segítségével létrehozunk egy token2 (szóköz nélkül) nevű változót, a kinyert számértékkel. A deklarálást megvalósító függvény visszatér egy sikert vagy hibát reprezentáló számértékkel, ami alapján állítódik a következő állapot. Ezzel a sor feldolgozásának vége.

A többi állapot sorainak feldolgozása is teljesen hasonló módon zajlik. A kifejezéseket az ExpressionParser modul értékeli ki, a token-eket szövegkezelő függvényekkel nyerjük ki.

3.5.5 - Feltételes elágazások és ciklusok

A futásidejű feltételes elágazások és ciklusok a forráskódban nem csak egy sorra terjednek ki.

Állapot neve	Rövid leírás
IF	If-else szerkezet feldolgozása.
ENDIF	Csak igaz IF állapot után futhat le.
ELSE	Csak hamis IF állapot után futhat le.
WHILE	While-szerkezet feldolgozása.
END_WHILE	Csak igaz WHILE után futhat le
ERROR_IF_NOEND	Lezáratlan if-szerkezet.
ERROR_WHILE_NOEND	Lezáratlan while-szerkezet.

3.5.5 – 1. ábra: Futásidejű feltételes elágazásokat és ciklusokat feldolgozó állapotok

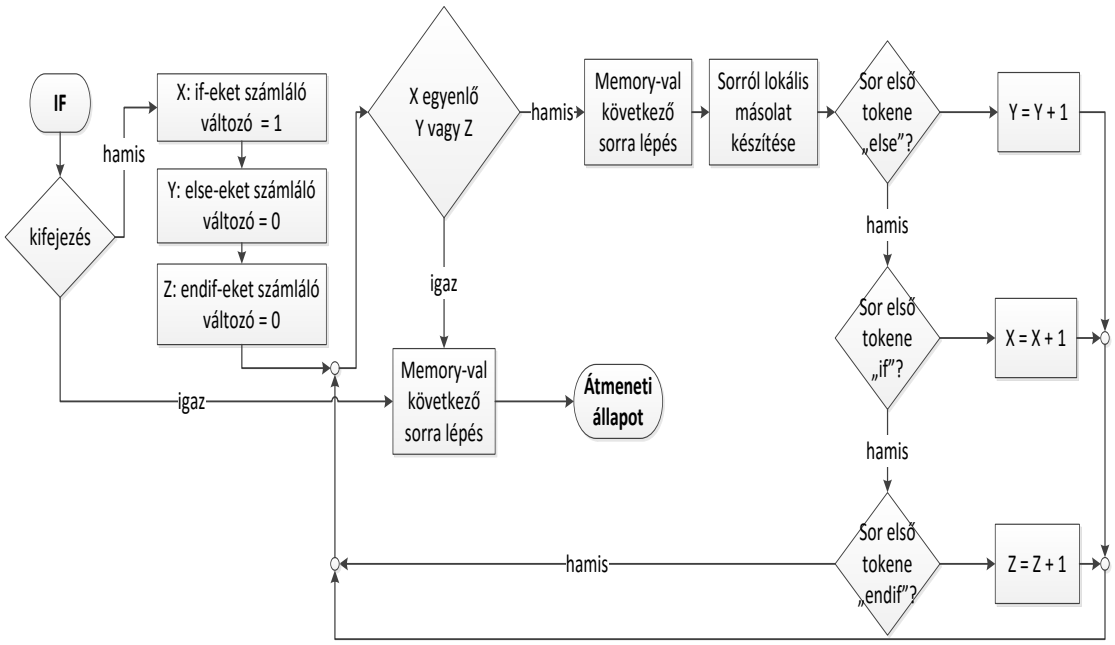
Az futásidejű feltételes elágazások és ciklusokat megvalósító állapotok abban térnek el a többi állapottól, hogy a sorok között lépkedhetnek.

```
if(kifejezés1)
…
    if(kifejezes2)

    …

    else

    …

       endif
…
endif
```

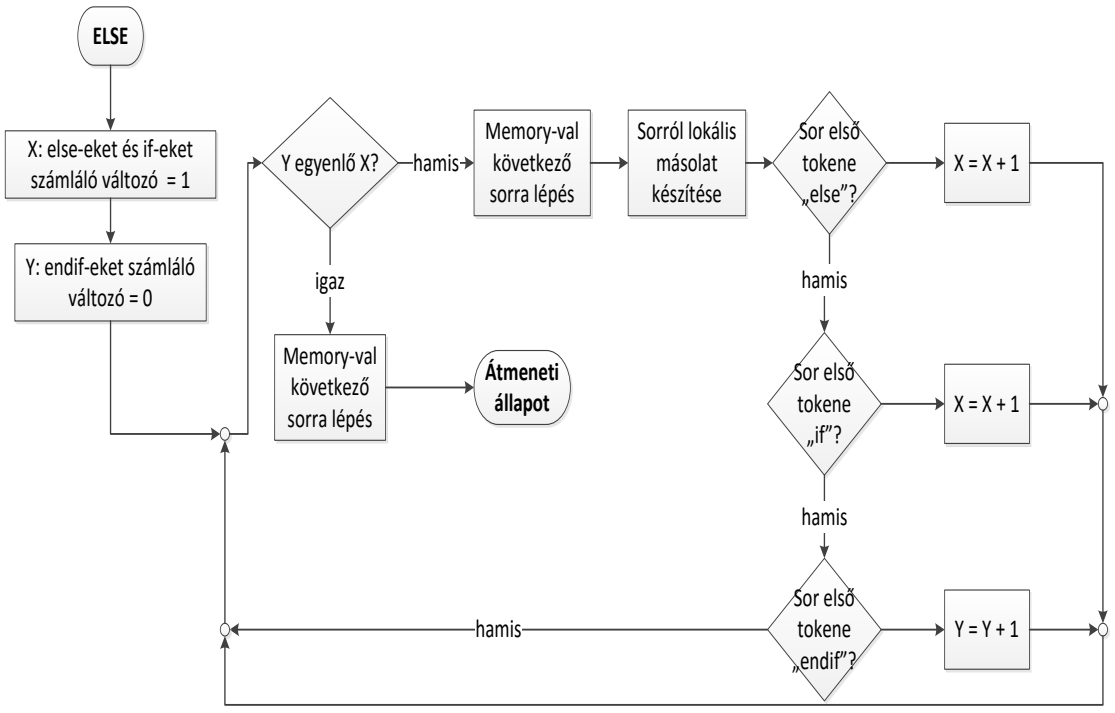
3.5.5 – 2. ábra: Egymásba ágyazott if-else szerkezet

Ha a kifejezés igaz, az állapotgép egyszerűen a következő sorra lép. Ha hamis az IF állapot a megfelelő sorokat kihagyja else vagy endif-ig, a záró sorral együtt. Ezért ELSE állapot csak igaz IF állapot után lehet.

A sorok feldolgozásának végén ellenőrizve van, hogy az if-ek és endif-ek száma egyenlő-e a forráskódban. Sorok közti lépésnél a feldolgozandó sorok számát tartalmazó változó értékét megfelelően állítjuk. IF állapot és az ELSE állapot feldolgozását a 3.5.5 – 3. és 4. ábra szemlélteti. A sorok közötti lépkedés a feltöltött sorok határai közé van szorítva.



3.5.5 – 3. ábra: IF állapot folyamatábrája



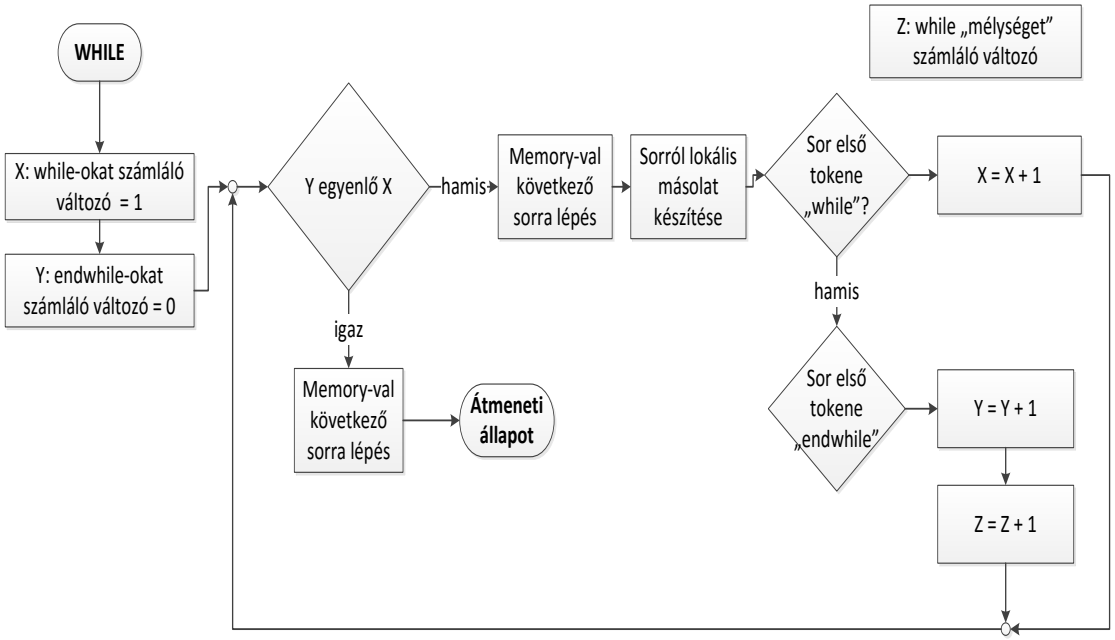
3.5.5 – 4. ábra: ELSE állapot folyamatábrája

A WHILE és az END_WHILE állapot feldolgozása is hasonlóan történik. Ha a kifejezés igaz, az állapotgép a következő sorra ugrik. Az END_WHILE állapot a sorok közt visszafelé lépkedve keresi a hozzátartozó nyitó while sort. Ha a kifejezés hamis, a megfelelő endwhile-ig kihagyjuk a sorokat. END_WHILE állapot csak igaz kifejezés után

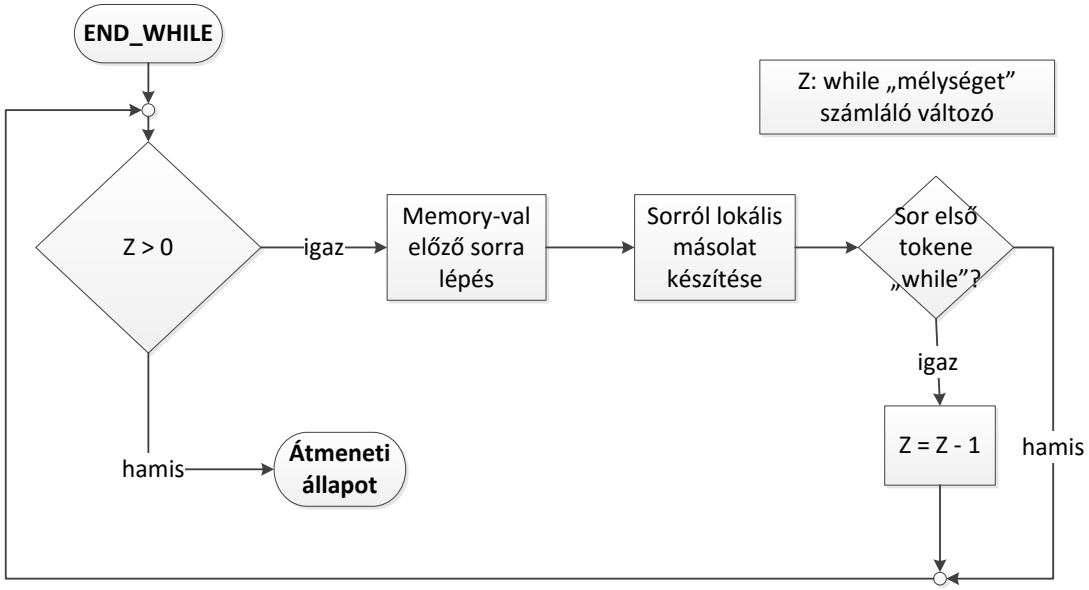
lehet. A while-szerkezetek „mélységét” egy program lefutása élettartamú, modulra lokális változóban számoljuk mindkét állapottal.

```
while(kifejezés1)
…
    while(kifejezés2)
    …
    …
       endwhile
…
endwhile
```

3.5.5 – 5. ábra: Egymásba ágyazott while-szerkezetek



3.5.5 – 6. ábra: WHILE állapot folyamatábrája



3.5.5 – 7. ábra: END_WHILE állapot folyamatábrája

3.6 - A kifejezés feldolgozó modul

A kifejezések feldolgozását az ExpressionParser modul valósítja meg. Erre több helyen is szükség van. Ha egy változónak akarunk értéket adni, vagy változókkal műveleteket végezni, kifejezéssel van dolgunk. Feltételes ciklusoknál, elágazásoknál, a feltételt ki kell tudnunk értékelni, hogy a további feldolgozásról dönteni tudjunk.

3.6.1 - Funkcionális specifikáció

Elvárások a modullal kapcsolatban:

- tizedespontos számábrázolás felismerése
- aritmetikai operátorok támogatása (+, -, *, /),
- bármilyen mély zárójelezés támogatása,
- relációs és logikai operátorok támogatása (==, !=, >, <, >=, <= , !, &&, ||),
- változók és visszatérési értékkel rendelkező C-ben implementált függvények támogatása.

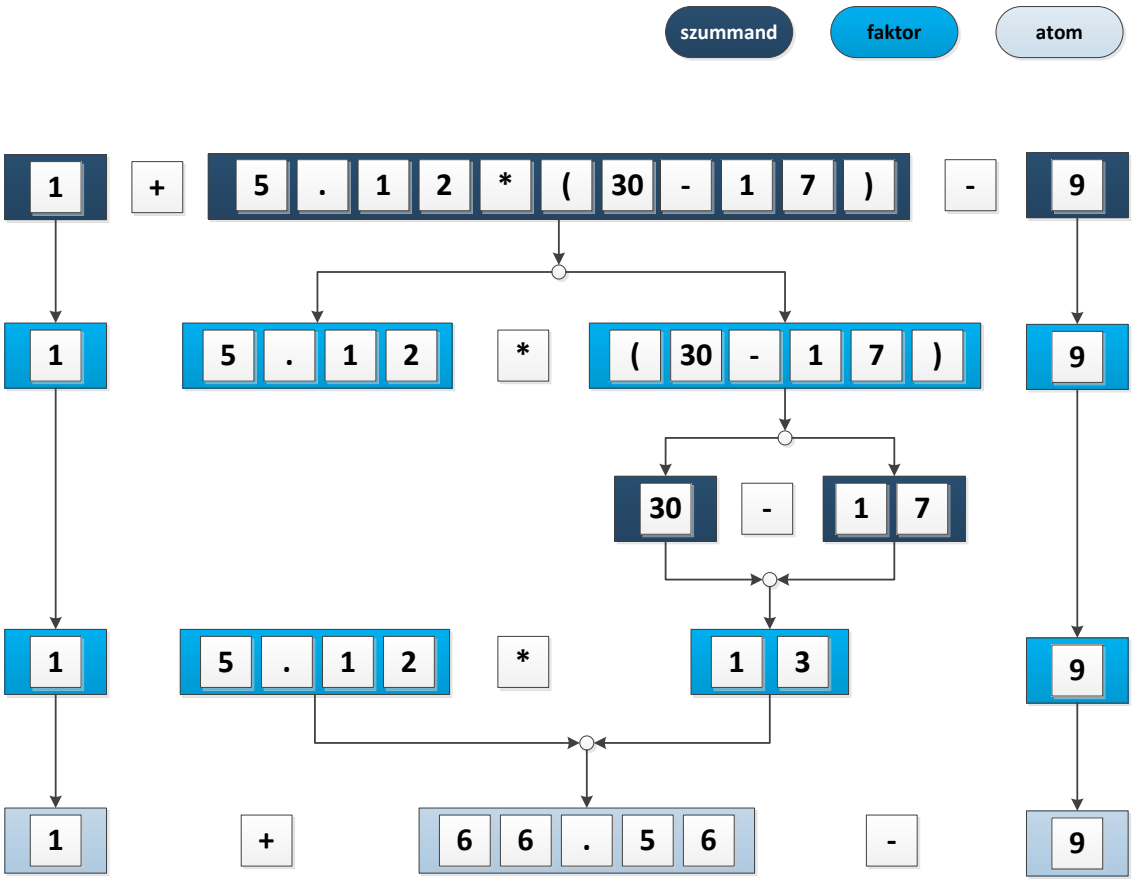
3.6.2 - Megvalósítás

A bemenet a kifejezés karaktereinek sorozata. A kimenet egy konkrét számérték. A feldolgozást karakterenként kell elvégezni. Figyelembe kell venni a műveletekből és zárójelezésből eredő, matematikai szabályok által diktált kiértékelési sorrendet is. Számos olyan eset előfordulhat, amikor az aktuális bemenet, azaz egy karakter alapján, nem tudunk dönteni a további feldolgozásról, mivel az függ a későbbi karakterektől. Pl.

a (2+2*7) kifejezés feldolgozása közben, a **+** operátornál tartva, a hozzáadandó érték függ a későbbi ***** operátortól. A (((2+2)*(2.5+10)) / 2) esete még bonyolultabb.

A bemenet feldolgozását az operátorokra alapozzuk, mivel ezek határozzák meg a műveletek sorrendjét, és a kifejezés részei közti relációkat.

- Nevezzük a +,- által elválasztott részeket *szummandoknak.*
- Nevezzük a szummandon belüli *,/ által elválasztott részeket *faktoroknak*.
- Nevezzük a faktorokat felépítő tagokat *atomoknak*.



3.6.2 – 1. ábra: Kifejezés felbontása

Egy szintaktust kapunk három elemmel. A (1.) szummandok képzik a legfelső szintet, (2.) faktorokból épülnek fel, a faktorok pedig (3.) atomokból.

A kifejezést az *alászálló értelmező* (angolul „descent parser”) módszerével értékeljük ki. Minden, a struktúrát felépítő elemhez egy-egy feldolgozóegységet hozunk létre:

1. a szummandok feldolgozását a ParseSummand() függvény végzi,
2. a faktorok feldolgozását a ParseFactor() függvény végzi,
3. az atomok feldolgozását a ParseAtom() függvény végzi.

A függvények a modulra lokálisak.

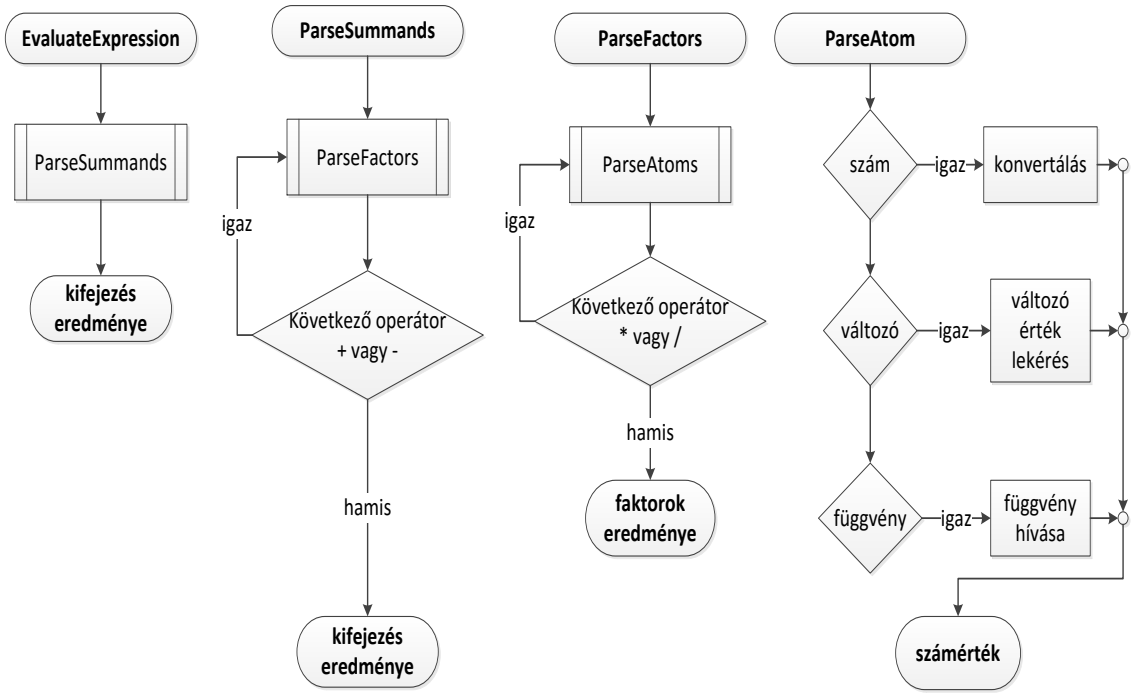
A faktornak két típusa van:

- atom,
- és zárójeles részkifejezés.

Az atom lehet:

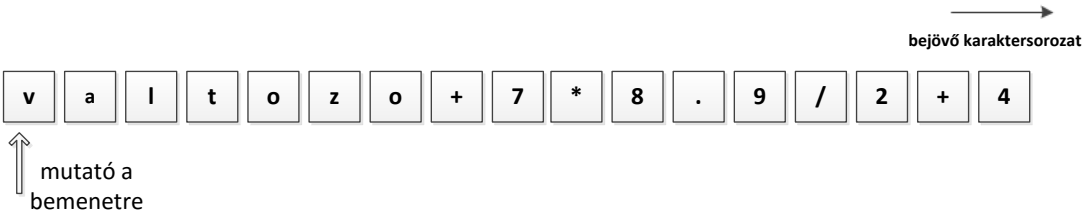
- egy konkrét számérték,
- változó,
- vagy egy C-ben implementált visszatérési értékkel rendelkező függvény.

A teljes kifejezés kiértékelését az EvaluateExpression() függvény végzi.



3.6.2 – 2. ábra: Feldolgozó függvények

A feldolgozóegységek csak a saját maguk által „leírt” elemeket dolgozzák fel, a kifejezést belülről kifelé, „alászállva” értelmezzük. A kifejezésben található nyitó és záró zárójelek azonos száma a kiértékelés a feldolgozás végén ellenőrizve van.



3.6.2 – 3. ábra: Bejövő karaktersorozat

A kifejezést karakterenként dolgozzuk fel. A kifejezés végét általunk definiált karakterek jelzik. Ezek főleg a mérésvezérlő forráskódban felhasznált helyekből erednek. Pl. tömb indexértékének számolása miatt ilyen a szögletes záró zárójel.

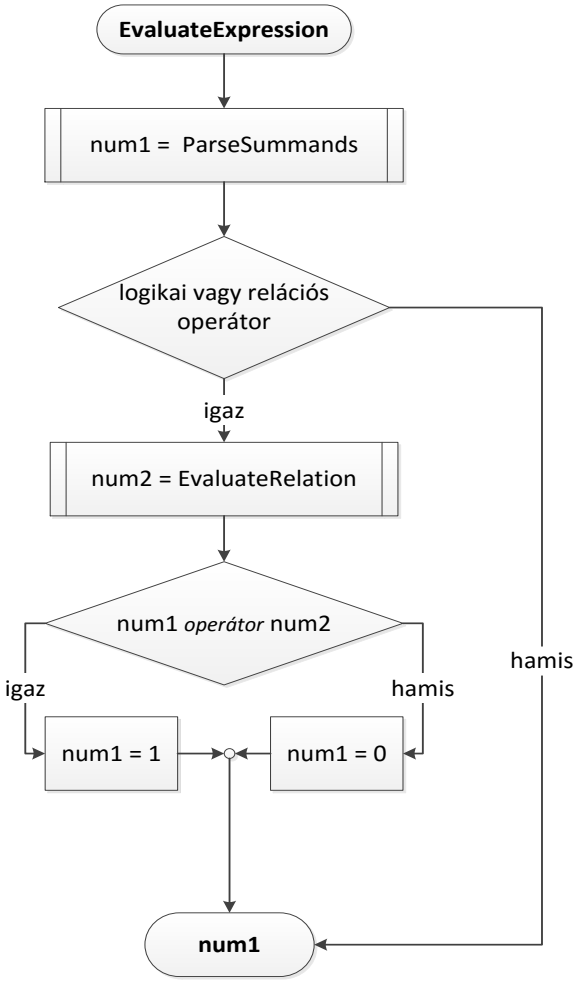
Az atomok lehetséges típusait adott karakterek alkotják.

Atom	Alkotó karakterek
konkrét számérték	számkarakterek és tizedespont
változó vagy függvény	angol abc betűi aláhúzás karakter számérték

3.6.2 – 4. ábra: Atomokat felépítő karakterek

A maximális név hosszúság általunk definiált. Ha *névalkotó karakter* a bemenet, a feldolgozást átveszi egy szövegrészlet mentő függvény. Ez addig másolja a karaktereket egy általunk létrehozott karaktertömbbe, amíg a bemenet megegyezik valamelyik névalkotó karakterrel. Az utolsóutáni karakter címével tér vissza, ennek segítségével folytatódik majd tovább a feldolgozás. Az elmentett névvel megnézzük van-e ilyen változó vagy függvény. Változó értékének lekérésére a Variables modul kínál függvényt. A mérésvezérlő forráskódban szereplő függvény neve, közvetve egy switch-case szerkezetet vezérelve, szövegkezelő függvényekkel kinyerve és számmá konvertálva a paramétereket, meghívja a megfelelő függvényt, és visszatér a számértékével. Ha az atom nem változó vagy függvény név, számértéknek kell lennie. Ezt egyszerűen számmá konvertáljuk, majd kiszámoljuk az utolsó utánikarakter címét a további feldolgozáshoz.

A logikai és relációs operátorok két kiszámolt kifejezés értékét hasonlítják össze. Ezt az EvaluateRelation() függvény végzi.



3.6.2 – 3. ábra: Reláció feldolgozása

3.7 - Sorok és változók tárolása

3.7.1 - Változókat kezelő modul

Modul leírása:

Külső változók kezelése.

- változó deklarálása a külső SRAM-on,
- tömb deklarálása a külső SRAM-on,
- tizedespontos számábrázolás felismerése,
- több karakter hosszú név támogatása,
- keresés név alapján,
- érték lekérése, beállítás név alapján.

A mérési eredmények, és a mérésvezérléséhez szükséges változók használatának lehetősége. A mérési vezérlésben deklarált változókat a külső SRAM-on tároljuk. Egy változóhoz két bejegyzés tartozik, a változó neve, és a változó értéke. Ez egy definiált méretű adatstruktúra. A mérési vezérlésben lehetőség van tömb deklarálására is, ennek megvalósítása egyszerűen a tömb nevével megegyező nevű, indexelt változók deklarálása. A modul figyeli a rendelkezésére álló helyet, ezt nem lépi túl.

Kínált funkciók:

- változó deklarálása,
- értékadás változónak,
- változó keresése név alapján.

3.7.2 - Memória modul

Modul leírása:

A mérésvezérlő forráskód soronkénti tárolása az SRAM-on.

Kínált funkciók:

- definiálható sor méret
- lépkedés a sorok között írás/olvasás céljából,
- sor írása/olvasása,
- rendelkezésre álló terület határainak figyelése,
- sorok számának tárolása.

4 - A megvalósítás tapasztalatai

A fejlesztést megelőző félévben C++ programnyelven programoztam. Az objektum orientált nyelvben elsajátított tervezési elvek és megoldások C nyelven is közel olyan jól alkalmazhatóak voltak, és úgy érzem nagyban hozzájárultak a fejlesztés előrehaladásához. A C nyelvben ezek átgondolása közelebbi ismeretekhez vitt a modularitás, változók és függvények láthatósága és élettartama, saját definiált adatstruktúrák, és feldolgozó algoritmusok témakörökben.

A jelenlegi megvalósításban a feldolgozó algoritmusok megtervezése nagyrészt szöveges alapú volt. Ez a kívánt feldolgozás bonyolultságából eredően nem volt túl hatékony. Rengetek próbálgatás, és több véletlent is követelt. Ha valamilyen formális nyelvel le tudtam volna írni a feldolgozást és annak menetét, a tervezés sokkal hatékonyabb lett volna. A megkívánt matematikai ismeretek elsajátítása számomra igen jó befektetésnek tűnik szoftveres szempontból.

A probléma jellegéből fakadóan mélyebb ismeretekre tettem szert arról, hogyan lesz az általunk írt forráskódból, a processzor számára fizikailag értelmezhető utasítássorozat. A mikrokontroller és a külső perifériák között megvalósított szabvány alapú kommunikáció szoftveres megvalósítása is a működés fizikai jellegére nyújtott rálátást. Ezek számos érdekes témát vetettek fel.

A hardveren felhasznált megoldások elmagyarázása, úgy gondolom rendkívül hasznos volt számomra, e területen én mindig úgy éreztem, nehéz a jó szakirodalom felkutatása.

Rengeteg olyan témát súroltam, amivel a fejlesztés folyamán nem volt elég időm foglalkozni, de fel lettek jegyezve, és a jövőben mindenképp tervezek. Emellett a C programozási nyelvről is rengeteg új tapasztalatra tettem szert.

5 - Felhasznált irodalom

1 - Bevezetés

1.1 - A témakör indoklása

(2008) Dr. Horváth Elek - Méréstechnika (BMF-KKVFK – 1161)

1.2 - A témakör felvezetése

(2013) http://en.wikipedia.org/wiki/Programming_languages(2013)
http://cplus.about.com/od/introductiontoprogramming/a/compinterp.htm
(2013) http://stackoverflow.com/questions/3618074/what-is-the-difference-between-compiler-and-interpreter
(2013) http://searchcio-midmarket.techtarget.com/definition/bytecode
(2013) http://en.wikipedia.org/wiki/Interpreter_%28computing%29
(2013) Microchip MPLAB User's Guide (DS515119A), - 1.1 An Overview of Embedded Systems
(2013) http://en.wikipedia.org/wiki/Bytecode
(2013) http://en.wikipedia.org/wiki/V8_%28JavaScript_engine%29

1.4 - Lehetséges megoldások

(2013)http://staff.ustc.edu.cn/~han/CS152CD/Content/COD3e/CDSections/CD2.12.pdf
(2013) http://hu.wikipedia.org/wiki/Ford%C3%ADt%C3%B3program
(2013) http://en.wikipedia.org/wiki/Compiler
(2013) http://en.wikipedia.org/wiki/Lexical_analysis
(2013) http://hu.wikipedia.org/wiki/Regul%C3%A1ris_kifejez%C3%A9s

1.5 - Választott megoldás

(2013) http://compilers.iecc.com/crenshaw/
(2013) http://www.elook.org/programming/c/stdstring.html

3 - Szoftver

3.1 - Megvalósítás módja

(2013) http://heim.ifi.uio.no/frank/inf5040/CBSE/Component-Based_Software_Engineering_-_ch1.pdf
(2013) http://stackoverflow.com/questions/1256009/how-important-is-modularization-of-software-projects
(2013) www.hccfl.edu/media/99380/chapter03.ppt

(2013) http://www.powershow.com/view/1e2017-MmIzN/Modularisation_II_powerpoint_ppt_presentation
(2013) http://stackoverflow.com/questions/212270/should-application-architects-write-code#212362
(2013)http://www.softwareresearch.net/fileadmin/src/docs/teaching/WS07/SaI/Mutinelli_Zwettler_slides.pdf
(2011) Dr. Schuster György - C programozási nyelv munkapéldány

3.5 - Az Interpreter modul

3.5.2 - Saját nyelv szintaktikája

(2013) http://www.cplusplus.com/files/tutorial.pdf

3.5.3 - Az állapotgép megvalósítása

(2013) https://en.wikipedia.org/wiki/Finite-state_machine

3.5.4 - Sorok feldolgozása

(2013) http://www.elook.org/programming/c/

3.6 - A kifejezés feldolgozó modul

(2013)http://infoc.eet.bme.hu/ea14.php#6
(2013) http://www.strchr.com/expression_evaluator

6 - Hivatkozások

[1] (2013) http://ww1.microchip.com/downloads/en/DeviceDoc/70594C.pdf
[2] (2013) http://ww1.microchip.com/downloads/en/DeviceDoc/41574A.pdf
[3] (2013) http://ww1.microchip.com/downloads/en/DeviceDoc/39632e.pdf
[4] (2013) http://www.microchip.com/pagehandler/en-us/family/mplabx/
[5] (2013)
http://www.microchip.com/stellent/idcplg?IdcService=SS_GET_PAGE&nodeId=1406&dDocName=en010065
[6] (2013)
http://www.microchip.com/stellent/idcplg?IdcService=SS_GET_PAGE&nodeId=1406&dDocName=en537580